MODÉLISATION DE LA SÛRETÉ DE FONCTIONNEMENT DE SYSTÈMES À PARTIR DU LANGAGE AADL

Ana-Elena Rugina, Karama Kanoun, Mohamed Kaâniche

MODÉLISATION DE LA SÛRETÉ DE FONCTIONNEMENT DE SYSTÈMES À PARTIR DU LANGAGE AADL
SYSTEM DEPENDABILITY MODELING USING AADL LANGUAGE[1]

Ana-Elena Rugina, Karama Kanoun et Mohamed Kaâniche
LAAS-CNRS
7, avenue du Colonel Roche – 31077 Toulouse Cedex 4

Résumé

Pour des raisons d'efficacité et de maîtrise des coûts, le souhait des concepteurs de systèmes est d'utiliser de façon intégrée un ensemble cohérent de formalismes pour décrire les spécifications et conceptions, et également pour effectuer des analyses de sûreté de fonctionnement. AADL (Architecture Analysis and Design Language) a prouvé son efficacité pour la modélisation d'architectures. Par conséquent, ce langage est de plus en plus utilisé dans des processus industriels d'ingénierie système. Cet article présente un cadre de modélisation permettant la génération de modèles analytiques de sûreté de fonctionnement à partir de modèles AADL dans l'objectif de faciliter l'obtention de mesures de sûreté de fonctionnement comme la fiabilité et la disponibilité. Nous proposons une approche itérative de modélisation. Le modèle AADL de sûreté de fonctionnement est transformé en un RdPSG (Réseau de Petri Stochastique Généralisé) en appliquant des règles de transformation de modèle. Le RdPSG résultant peut être traité par des outils existants. L'approche est illustrée sur un petit exemple.

Summary

For efficiency and cost control reasons, system designers' will is to use an integrated set of methods and tools to describe specifications and designs, and also to perform dependability analyses. AADL (Architecture Analysis and Design Language) has proved to be efficient for architecture modeling. Consequently, it is more and more used in industrial system engineering processes. This paper presents a modeling framework allowing the generation of dependability-oriented analytical models from AADL models, to facilitate the evaluation of dependability measures, such as reliability or availability. We propose a stepwise approach for system dependability modeling using AADL. The AADL dependability model is transformed into a GSPN (Generalized Stochastic Petri Net) by applying model transformation rules. The resulting GSPN can be processed by existing tools. The modeling approach is illustrated on a small example.

1 Introduction

La complexité croissante des systèmes informatiques critiques entraîne des difficultés d'ingénierie système, en particulier liées à la validation et à l'analyse des performances et des exigences concernant la sûreté de fonctionnement. Des approches d'ingénierie à base de langages de description d'architectures sont de plus en plus utilisées dans l'industrie dans le but de maîtriser cette complexité au niveau de la conception. En particulier, AADL (Architecture Analysis and Design Language) [1] a fait l'objet d'un intérêt croissant pendant ces dernières années. AADL a été développé et récemment standardisé par la « International Society of Automotive Engineers » (SAE), pour faciliter la conception et l'analyse de systèmes complexes, critiques, temps réel dans des domaines comme l'avionique, l'automobile et le spatial. AADL fournit une notation textuelle et graphique standardisée pour décrire des architectures matérielles et logicielles et pour effectuer différentes analyses pour déterminer le comportement et la performance du système modélisé. Le langage a été conçu pour être extensible afin de permettre la réalisation d'analyses qui ne sont pas réalisables avec le langage de base.

En plus de la description du comportement du système en présence de fautes, les développeurs sont intéressés par l'obtention de mesures quantitatives d'attributs de la sûreté de fonctionnement, pertinents pour leurs systèmes, comme la fiabilité et la disponibilité. Pour des raisons pragmatiques, les concepteurs de systèmes qui utilisent une approche d'ingénierie basée sur AADL sont intéressés d'avoir un ensemble intégré de méthodes et outils pour décrire les spécifications et conceptions et pour effectuer des évaluations de la sûreté de fonctionnement. Dans cette optique, une annexe au standard AADL a été définie récemment (« *AADL Error Model Annex* » [2]) pour compléter les capacités de description du langage de base. Cette annexe représente un sous langage avec une sémantique précise qui sert à décrire les caractéristiques du système modélisé en AADL liées à la sûreté de fonctionnement (fautes, modes de défaillance, hypothèses de réparation, propagations d'erreurs, etc.). Cependant, à l'état actuel, il n'y a aucune méthodologie pour aider les développeurs dans l'utilisation des notations proposées pour décrire des modèles de sûreté de fonctionnement complexes qui reflètent des systèmes réels, avec de multiples interactions et dépendances entre les composants. L'un des deux objectifs de cet article est de proposer une méthode structurée pour faciliter l'élaboration de modèles AADL de sûreté de fonctionnement.

Le « AADL Error Model Annex » mentionne qu'il est possible de générer des automates stochastiques comme des arbres de fautes et des chaînes de Markov à partir de modèles AADL enrichis avec des informations liées à la sûreté de fonctionnement. En effet, les chaînes de Markov sont reconnues pour être des moyens puissants pour la modélisation de la sûreté de fonctionnement des systèmes en prenant en compte les dépendances entre les composants. Habituellement, les réseaux de Petri stochastiques généralisés (RdPSG) sont utilisés pour générer automatiquement des chaînes de Markov. De plus, les RdPSG permettent de vérifier structurellement le modèle avant la génération de la chaîne de Markov. Ces facilités pour la vérification sont très utiles quand on traite de grands modèles. Durant les dix dernières années, différentes approches, basées sur les RdPSG et leurs extensions, ont été définies pour maîtriser la construction et la validation de modèles de sûreté de fonctionnement (voir e.g. [3-5]). La méthode de modélisation proposée dans cet article vise à profiter de telles approches dans le contexte d'un processus d'ingénierie système basé sur AADL pour i) construire le modèle AADL de sûreté de fonctionnement et pour ii) générer des modèles RdPSG de sûreté de fonctionnement à partir de modèles AADL par transformation de modèle. De cette manière, la complexité de la génération du modèle RdPSG est masquée aux utilisateurs qui connaissent AADL et qui ont des connaissances limitées dans le domaine des RdPSG. Les modèles AADL et RdPSG sont construits de manière itérative, en prenant en compte progressivement les dépendances entre les composants. Les modèles sont également validés à chaque itération.

Pour récapituler, cet article a deux objectifs : i) proposer une méthode structurée pour guider l'élaboration du modèle AADL de sûreté de fonctionnement et ii) présenter des exemples de règles de transformation de modèle pour générer des RdPSG à partir de modèles AADL de sûreté de fonctionnement. L'ensemble des règles de transformation de modèle est conçu pour être mis en œuvre dans un outil de transformation de modèle, de façon transparente à l'utilisateur. Un tel outil peut être interfacé avec l'un des outils existants de traitement de RdPSG (e.g., Surf-2 [6], Möbius [7], Sharpe [8], GreatSPN [9], SPNP [10]) pour évaluer les mesures de sûreté de fonctionnement et de performabilité.

La suite de cet article est organisée de la manière suivante. La section 2 discute de l'état de l'art . La section 3 présente les concepts de AADL nécessaires pour comprendre notre approche de modélisation. La section 4 est une vue d'ensemble de notre approche itérative de modélisation basée sur AADL. La section 5 présente des exemples de

[1] Ce travail a été partiellement financé par 1) la Commission Européenne (projet européen intégré ASSERT No. IST 004033 – www.assert-online.net et réseau d'excellence ReSIST No. IST 026764 – www.laas.fr/RESIST/) et 2) le Fond Social Européen.

règles de transformation de AADL vers RDPSG. La section 6 illustre notre approche sur un petit exemple et la section 7 conclut l'article.

2 Travaux connexes

À notre connaissance, il n'y a pas de contribution similaire à la nôtre dans l'état de l'art actuel. La plupart des articles publiés concernant des analyses basées sur AADL se sont concentrés sur l'extension du langage pour faciliter la vérification formelle. Par exemple, le projet COTRE [11] a fourni une approche de conception reliant la vérification formelle et les exigences exprimées en langages de description d'architecture. Des spécifications de système en AADL peuvent être importées dans le nouveau langage COTRE. Une spécification en langage COTRE peut être transformée en un automate temporisé, réseau de Petri temporel ou en un autre modèle analytique. Cependant, à notre connaissance, il n'y a pas de contribution visant l'obtention de modèles d'évaluation quantitative de la sûreté de fonctionnement à partir de spécifications en langage COTRE.

En considérant le problème de la génération de modèles d'évaluation de la sûreté de fonctionnement à partir de langages utilisés dans les approches d'ingénierie système à base de modèles dans un contexte plus large, plusieurs travaux de recherche ont été menés autour de UML (Unified Modeling Language) [12]. Par exemple, le projet européen HIDE ([13], [14]) a proposé une méthode pour analyser et évaluer automatiquement la sûreté de fonctionnement à partir de modèles UML. Dans ce projet, plusieurs transformations de modèle ont été définies : i) à partir de diagrammes UML structurels et comportementaux vers des RdPSG, des réseaux de Petri déterministes et stochastiques, et des réseaux stochastiques à base de récompenses (« Stochastic Reward Nets ») pour l'évaluation de mesures de sûreté de fonctionnement, ii) à partir de diagrammes UML « statecharts » vers des structures de Kripke pour la vérification formelle et iii) à partir de diagrammes UML de séquence vers des réseaux stochastiques à base de récompenses pour des analyses de performance. Aussi, [15] propose un algorithme pour obtenir des arbres de fautes dynamiques à partir de modèles UML du système (un ensemble de diagrammes de classe, objet et déploiement, étendus avec des stéréotypes et « tagged values »).

AADL est différent de UML, car en AADL l'utilisateur manipule un seul modèle d'architecture annoté. Par conséquent, les approches de modélisation mentionnées plus haut ne peuvent pas être directement appliquées dans le contexte de AADL. Le cadre de modélisation présenté dans cet article est complémentaire aux travaux mentionnés plus haut et il a pour objectif d'assurer une meilleure intégration des techniques d'évaluation de la sûreté de fonctionnement basées sur RdPSG dans des approches d'ingénierie système basées sur AADL.

3 Concepts de AADL

Le langage AADL de base permet d'analyser l'impact de différents choix architecturaux (comme la politique d'ordonnancement ou la redondance) sur les propriétés du système [16]. Une spécification d'architecture en AADL décrit comment les composants sont combinés en sous-systèmes et comment ils interagissent. Les architectures sont décrites de manière hiérarchique. Les *composants* sont les briques de base des architectures AADL. Ils sont groupés en trois catégories : 1) logicielle (processus, sous-programme, donnée, fil d'exécution « `thread` », groupe de fils d'exécution), 2) matérielle (processeur, mémoire, dispositif, bus) et 3) composée (système). Les composants AADL peuvent être composés de sous-composants et interconnectés à travers des caractéristiques (« features ») comme les ports, appels de sous-programmes et paramètres. Ces caractéristiques spécifient comment les composants s'interfacent entre eux. Chaque composant AADL a deux niveaux de description : le *type* et l'*implémentation*. Le type décrit comment l'environnement « voit » ce composant (e.g. ses propriétés et caractéristiques). Une ou plusieurs implémentations peuvent être associées au même type, correspondant à différentes structures du composant en termes de sous-composants, connexions, appels de sous-programmes et modes opérationnels.

Le langage AADL de base est conçu pour décrire des architectures statiques avec des modes opérationnels pour leurs composants. Néanmoins, le langage de base peut être étendu afin de permettre à l'utilisateur d'associer des informations supplémentaires à l'architecture. Les *modèles d'erreur AADL* sont une extension qui a pour objectif de permettre la réalisation d'analyses (qualitatives et quantitatives) de sûreté de fonctionnement. Le « AADL Error Model Annex » définit un sous langage qui permet de déclarer des modèles d'erreur dans une librairie. Le modèle AADL d'architecture sert de squelette aux modèles d'erreur car ils doivent être instanciés et associés aux composants.

Les modèles d'erreur des composants décrivent le comportement des composants auxquels ils sont associés en présence i) de fautes et événements de réparation et ii) de propagations d'erreurs de l'environnement. Comme pour un composant du langage AADL de base, un modèle d'erreur est spécifié sous forme d'un type et d'une ou plusieurs implémentations appropriées pour la réalisation de différentes analyses de sûreté de fonctionnement. Le type déclare des états (`error states`), des événements internes au composant (`error events`) et des propagations (`error propagations`[2]) qui circulent à travers de connexions et liaisons du modèle d'architecture. De plus, l'utilisateur peut déclarer des propriétés de type « `Guard` » pour contrôler les propagations. Les implémentations déclarent des `transitions` entre les états et des propriétés stochastiques d'`Occurrence` pour les événements et les propagations sortantes. Les transitions sont déclenchées par des événements et propagations déclarés dans le type. Les propriétés d'Occurrence spécifient le taux d'arrivée ou la probabilité d'occurrence pour les événements et propagations. La Figure 1 montre un modèle d'erreur de base (sans propagations).

Figure 1 : Modèle d'erreur de base

Error Model Type [basic]

```
error model basic
features
Error_Free: initial error state;
Failed: error state;
Fail : error event {Occurrence => Poisson λ};
Repair: error event {Occurrence => Poisson μ};
end basic;
```

Error Model Implementation [basic.nominal]

```
error model implementation basic.nominal
transitions
Error_Free-[Fail] -> Failed;
Failed-[Repair] -> Error_Free;
end basic.nominal;
```

Les propagations sont directionnelles. L'identifiant `in` identifie les propagations entrantes, tandis que l'identifiant `out` identifie les propagations sortantes. Une propagation `out` se produit dans un modèle d'erreur source selon une propriété d'`Occurrence` spécifiée par l'utilisateur. Le modèle d'erreur source envoie la propagation à travers tous les ports et liaisons du composant auquel le modèle d'erreur est associé. Par conséquent, une propagation `out` arrive à un ou plusieurs modèles d'erreurs associés à des composants récepteurs. Si un modèle d'erreur récepteur déclare une propagation `in` avec le même nom que la propagation `out` reçue, la propagation `in` peut influencer son comportement (e.g. la propagation `in` pourrait déclencher des transitions entre des états et/ou des modes opérationnels). Tous les modèles d'erreur récepteurs d'une propagation sont influencés simultanément.

Des propriétés de type `Guard` (associées aux ports, données partagées et sous-programmes client-serveur) permettent le contrôle des propagations par des expressions booléennes. Par exemple, la propriété `Guard_In` définit des expressions booléennes qui spécifient comment les propagations arrivant à un composant sont filtrées ou masquées.

[2] Dans le AADL Error Model Annex, les termes état, événement et propagation sont associés au terme « **erreur** » pour mettre en évidence qu'ils sont spécifiés dans le AADL **Error** Model Annex. Cependant, les états peuvent représenter des états de bon fonctionnement, les événements peuvent représenter des réparations et les propagations peuvent représenter n'importe quelle notification. Dans la suite de l'article, nous omettrons le terme « erreur ».

Le modèle d'erreur du système est défini comme une composition d'un ensemble d'automates finis stochastiques correspondant aux composants. Comme le modèle d'architecture, le modèle d'erreur du système est décrit de manière hiérarchique. Si un composant et ses sous-composants ont des modèles d'erreur, alors la relation entre ces modèles doit être définie. L'utilisateur peut décrire l'état d'un composant comme i) une *fonction* des états de ses sous-composants (e.g. le modèle d'erreur du composant est dérivé des modèles d'erreur des sous-composants) ou comme ii) une *abstraction* du comportement de ses sous-composants. La relation entre ce type d'abstractions et les modèles d'erreur concrets est explorée dans [17].

4 L'approche de modélisation

Pour des systèmes complexes, la principale difficulté dans la construction du modèle de sûreté de fonctionnement est due aux dépendances entre les composants du système. Les dépendances peuvent être de plusieurs types, identifiés dans [4] : structurelles, fonctionnelles ou liées à la tolérance aux fautes et à la stratégie de maintenance. Comme le comportement des composants peut dépendre des comportements d'autres composants, les utilisateurs ont besoin d'une approche structurée pour modéliser systématiquement les dépendances. Une telle approche aide à l'évitement des erreurs dans le modèle du système et facilite sa validation. Dans notre approche, le modèle AADL de sûreté de fonctionnement est construit progressivement et de manière itérative. Plus concrètement, dans une première itération, nous proposons de modéliser les comportements des composants du système en présence de leurs propres fautes et événements de réparation. Par conséquent, les composants sont modélisés comme s'ils étaient *isolés* du reste du système. Dans les itérations suivantes, nous proposons de compléter le modèle en introduisant une par une les dépendances entre les composants.

Une vue d'ensemble de notre approche de modélisation, composée de quatre étapes, est illustrée par la Figure 2.

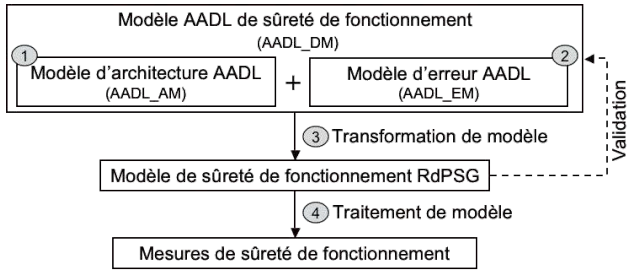


Figure 2 : Approche proposée

La première étape est consacrée à la modélisation de l'architecture du système en AADL (e.g. sa structure en termes de composants et modes opérationnels de ces composants). Le modèle d'architecture AADL (AADL_AM) peut être disponible à ce stade s'il a déjà été construit pour d'autres analyses.

La seconde étape est dédiée à la modélisation du comportement du système en présence de fautes et en tenant compte des interactions entre composants, par l'association de modèles d'erreur (AADL_EMs) aux composants du AADL_AM. L'ensemble de AADL_EMs associés aux composants forme le AADL_AM du système.

L'ensemble AADL_AM et AADL_EM forment un modèle de sûreté de fonctionnement du système en AADL (AADL_DM).

La troisième étape vise à construire un modèle de sûreté de fonctionnement sous forme de RdPSG à partir du AADL_DM à l'aide de règles de transformation de modèle. Cette étape inclut la validation sémantique du modèle.

La **quatrième étape** est dédiée au traitement du modèle RdPSG afin d'obtenir des mesures de sûreté de fonctionnement. Cette dernière étape est entièrement basée sur des algorithmes classiques de traitement des modèles RdPSG. Ne faisant pas l'objet de notre travail, cette étape n'est pas détaillée dans cet article.

Pour obtenir le AADL_DM, l'utilisateur doit effectuer la première et la deuxième étapes décrites ci-dessus. La troisième étape est conçue pour être automatisée afin de masquer la complexité de la génération du RdPSG. L'approche itérative peut être appliquée uniquement à la deuxième étape ou à la deuxième et troisième étapes ensemble. Dans le dernier cas, il est possible d'effectuer des vérifications structurelles et sémantiques basées sur le RdPSG après chaque itération, ce qui peut aider à identifier des erreurs de spécification dans le AADL_DM. Dans la suite de la section, nous donnons d'abord des règles de construction du AADL_DM et ensuite une vue d'ensemble de la transformation de AADL vers RdPSG.

4.1 La construction du modèle AADL de sûreté de fonctionnement

Le AADL_EM du système est construit en plusieurs itérations. Dans la première itération, nous construisons les AADL_EMs de base, associés aux composants, modélisant leurs comportements en présence de leurs propres fautes et événements de réparation. Dans les itérations suivantes, les dépendances entre les AADL_EMs sont modélisées progressivement. Les dépendances purement structurelles et fonctionnelles doivent être incluses dans le modèle avant les dépendances liées à la tolérance aux fautes et à la maintenance. Les dépendances liées à la tolérance aux fautes et à la maintenance peuvent avoir un impact sur la structure du système. En dernier, l'utilisateur peut définir des AADL_EMs dérivés pour des composants qui contiennent des sous-composants. Ainsi, le modèle final représente le comportement de chaque composant en présence de ses propres fautes et événements de réparation et en présence de fautes et événements de réparation dans des composants avec lesquels il interagit.

Il est à noter que tous les détails du AADL_AM ne sont pas nécessaires pour le AADL_DM. Seuls les composants ayant des AADL_EMs associés et les connexions / liaisons entre eux sont nécessaires.

Pour illustrer l'approche proposée, la suite de cette section présente successivement des règles pour modéliser i) une dépendance basée sur l'architecture (structurelle, fonctionnelle, ou liée à la tolérance aux fautes), ii) une dépendance liée à la maintenance et iii) un système hiérarchique dépendant de ses sous-composants.

Dépendance basée sur l'architecture

La dépendance est modélisée dans des AADL_EMs associés aux composants dépendants, en spécifiant respectivement des propagations sortantes (`out`) et entrantes (`in`) et leur impact sur le AADL_EM correspondant. Un exemple est donné dans la Figure 3 : *Composant 1* envoie des données au *Composant 2*, ainsi nous supposons que, au niveau du AADL_EM, le comportement du *Composant 2* dépend du comportement du *Composant 1*.

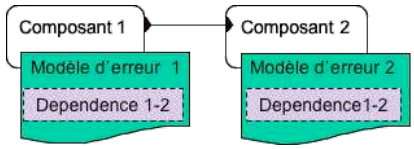


Figure 3 : Dépendance basée sur l'architecture

Supposons que le AADL_EM de la Figure 1 est associé au *Composant 1* et au *Composant 2* pour modéliser le comportement de chacun de ces deux composants comme s'ils étaient isolés. Pour modéliser une dépendance du *Composant 1* vers le *Composant 2*, nous devons ajouter :

- Dans le AADL_EM associé au *Composant 1* : i) la déclaration d'une propagation `out` et de sa propriété d'`Occurrence` dans le type et ii) une transition AADL déclenchée par cette propagation dans l'implémentation.
- Dans le AADL_EM associé au *Composant 2* : i) la déclaration de la propagation `in` correspondante dans le type et ii) une transition AADL déclenchée par elle dans l'implémentation.

Dépendance liée à la maintenance

Les dépendances liées à la maintenance doivent être décrites quand les dispositifs de réparation sont partagés entre les composants ou quand l'activité de réparation d'un sous-ensemble de composants doit être effectuée dans un ordre donné ou selon une stratégie spécifique (e.g. le logiciel ne peut être redémarré que si le matériel est disponible).

Des composants qui ne sont pas dépendants au niveau architectural peuvent le devenir à cause de la stratégie de maintenance. Par

conséquent, le AADL_AM pourrait nécessiter des ajustements pour supporter la description des dépendances liées à la stratégie de maintenance. Comme les AADL_EMs interagissent seulement par des propagations qui passent par des éléments architecturaux (e.g. connexions, liaisons), la dépendance liée à la maintenance doit avoir un support architectural. En d'autres termes, à part les composants de l'architecture, nous pouvons être conduits à ajouter un composant dans le AADL_AM pour décrire la stratégie de maintenance. La Figure 4-a montre un exemple de AADL_DM. Dans cette architecture, *Composant 3* et *Composant 4* n'interagissent pas au niveau de l'architecture AADL car il n'y a pas de dépendance basée sur l'architecture entre eux. Cependant, si nous supposons que les deux composants partagent un réparateur, la stratégie de maintenance doit être prise en compte dans le AADL_EM du système. Par conséquent, il est nécessaire de représenter le réparateur au niveau du AADL_AM, comme montré dans la Figure 4-b, pour modéliser explicitement la dépendance liée à la maintenance entre *Composant 3* et *Composant 4*. La partie du modèle qui correspond au réparateur est représentée en ligne pointillée puisqu'elle n'est utile que du point de vue du modèle de sûreté de fonctionnement.

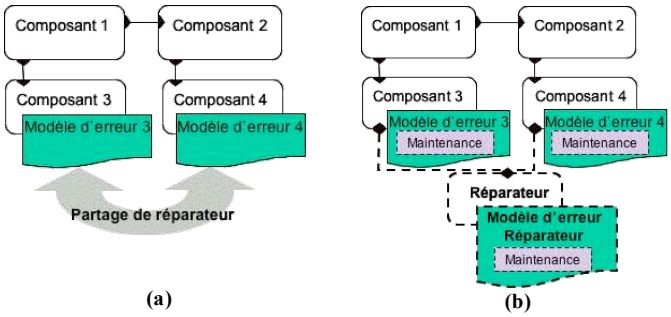


Figure 4 : Dépendance liée à la maintenance

Systèmes hiérarchiques

La Figure 5 montre un exemple de système hiérarchique où *Composant 1* est un composant qui contient des sous-composants et le AADL_EM qui lui est associé est *dérivé* des AADL_EMs de ses sous-composants. Dans ce cas, les états du *Composant 1* sont définis par une fonction des états de ses sous-composants, spécifiée par l'utilisateur.

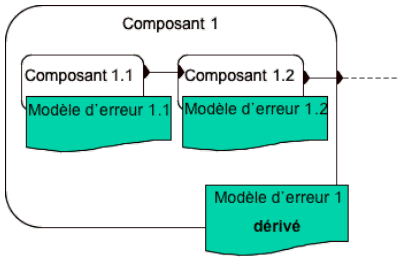


Figure 5 : Système hiérarchique

Le AADL Error Model Annex donne la possibilité de déclarer des AADL_EMs abstraits pour les composants qui contiennent des sous-composants. Cette option peut être utile pour faire abstraction des détails du modèle si un AADL_AM avec des AADL_EMs trop détaillés existe déjà. Si un AADL_EM abstrait est déclaré pour un composant, le composant est vu comme une boîte noire (e.g. les AADL_EMs des sous-composants ne font pas partie du AADL_DM).

4.2 Transformation de modèle (AADL vers RdPSG)

Le RdPSG obtenu après la transformation de modèle est formé de plusieurs blocs connectés par des arcs. Un bloc est un sous-réseau décrivant soit le comportement d'un composant en présence de ses propres fautes et événements de réparation (réseau composant), soit une dépendance (réseau dépendance). Dans le AADL_DM, chaque dépendance est modélisée dans les AADL_EMs concernées par la dépendance. Les réseaux dépendance sont obtenus à partir d'informations, concernant une dépendance, présentes dans (au moins) deux AADL_EMs dépendants. Le RdPSG global contient un réseau composant pour chaque composant et un réseau dépendance pour chaque dépendance. Un composant ayant un AADL_EM dérivé est transformé comme suit :

- Chaque sous-composant ayant un AADL_EM est transformé en un réseau composant,
- L'expression booléenne est transformée en un réseau de dépendance dérivé, car les états du composant dépendent seulement des états des sous-composants.

La Figure 6 montre le RdPSG modulaire obtenu par transformation à partir du AADL_DM de la Figure 3.

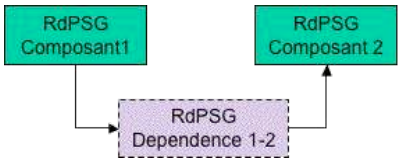


Figure 6 : RdPSG - représentation par blocs

La dépendance entre *Composant 1* et *Composant 2* est représentée par un bloc séparé. Les flèches qui relient les blocs représentent le sens de la dépendance. Ici, le sens est le même que le sens de la connexion dans le AADL_EM. Dans un cas plus général, quand les connexions sont bidirectionnelles, la dépendance au niveau du AADL_EM peut être uni- ou bidirectionnelle selon la direction explicite des propagations.

La structure modulaire du RdPSG permet la validation progressive du modèle. Le RdPSG est enrichi avec un nouveau bloc à chaque itération de la deuxième étape, c'est-à-dire quand une nouvelle dépendance est ajoutée au AADL_EM du système. Si un problème est détecté à l'itération *i*, seule la partie du modèle correspondant à l'itération *i* est remise en question.

5 Règles de transformation

Dans les trois sections suivantes nous présentons successivement des règles de transformation pour i) des composants isolés, ii) un ensemble de composants dépendants et iii) des systèmes hiérarchiques. Toutes les règles de transformation sont définies pour assurer par construction les propriétés syntaxiques du RdPSG. Les règles sont systématiques pour préparer l'automatisation de la transformation. Le RdPSG résultant est indépendant des outils. Néanmoins, les règles de transformation peuvent être simplifiées pour cibler certains outils évolués de traitement de RdPSG. Notons que dans la suite nous présentons seulement des exemples de règles de transformation. Un ensemble plus complet de règles est présenté dans [18].

5.1 Composants isolés

Dans le cas d'un composant isolé ou dans le cas d'un ensemble de composants indépendants, la transformation est plutôt directe, car un AADL_EM représente un automate stochastique, comme montré dans l'exemple de la Figure 1. Le Tableau 1 montre les règles de transformation de base. Le bloc RdPSG correspondant au composant est formé de places et de transitions. Le nombre de jetons dans un réseau composant est toujours 1, car à un moment donné un composant peut être dans un seul état.

5.2 Ensemble de composants dépendants

Les dépendances entre composants sont exprimées en AADL (voir Section 3) par : i) des propagations avec des noms correspondants et ii) des propriétés de type « `Guard` » (des mécanismes de contrôle des propagations). Pour des raisons didactiques, dans cette section, nous nous concentrons seulement sur la transformation pour des propagations avec des noms correspondants. Nous présentons d'abord un exemple d'une paire de AADL_EMs qui déclarent de telles propagations dans la Figure 7. Ensuite, nous illustrons deux transformations possibles sur cet exemple. Finalement, nous généralisons et discutons les avantages de chacune des deux règles.

Tableau 1 : Règles de transformation de base

Elément du AADL_EM	Elément du RdPSG	
Etat	Place	
Etat initial	Jeton dans la place correspondante	
Evénement	Transition RdPSG (temporisée ou immédiate)	
Propriété d'Occurrence d'un événement	Distribution ou probabilité caractérisant l'occurrence de la transition RdPSG associée	Temporisé Immédiat
Transition AADL (Etat_Source-[Evénement] -> Etat_Destination)	Arcs connectant des places (correspondant aux Etat_Source et Etat_Destination AADL) via transition RdPSG (correspondant à l'Evénement AADL)	

Figure 7 : Émetteur et récepteur – propagations avec noms correspondants

Error Model Type [Sender_example]

```
error model Sender_example
features
Error_Free: initial error state;
Failed: error state;
Fail: error event
              {Occurrence => poisson λ};
Sender_Failed: out error propagation
              {Occurrence => fixed p};
end Sender_example;
```

Error Model Implementation [Sender_example.basic]

```
error model implementation Sender_example.basic
transitions
Error_Free- [Fail] -> Failed;
Failed- [out Sender_Failed] -> Failed;
end Sender_example.basic;
```

Error Model Type [Receiver_example]

```
error model Receiver_example
features
Error_Free: initial error state;
Failed: error state;
Fail: error event {Occurrence => fixed p};
Sender_Failed: in error propagation;
end Receiver_example;
```

Error Model Implementation [Receiver_example.basic]

```
error model implementation
Receiver_example.basic
transitions
Error_Free- [Fail] -> Failed;
Error_Free- [in Sender_Failed] -> Failed;
end Receiver_example.basic;
```

Nous supposons que les deux AADL_EMs de la Figure 7 sont associés à des composants AADL qui sont connectés ou liés.

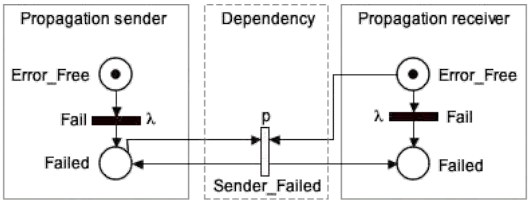
- Le AADL_EM à gauche correspond à l'émetteur de propagations. Il déclare deux états : *Error_Free* (état initial) et *Failed*. L'occurrence de l'événement *Fail* déclenche la transition AADL entre l'état source *Error_Free* et l'état destination *Failed*. Cet AADL_EM déclare également une propagation `out`, nommée *Sender_Failed*. Cette notification est propagée de l'état *Failed* (avec une probabilité *p*) et n'affecte pas l'état de l'émetteur.
- Le AADL_EM à droite correspond au récepteur de propagations. Il est identique à l'AADL_EM émetteur à l'exception du sens de la propagation *Sender_Failed*, qui est ici une propagation `in`. Si le récepteur reçoit la propagation *Sender_Failed* tout en étant dans l'état *Error_Free*, il passe dans l'état *Failed*.

Quand la propagation *Sender_Failed* se produit dans le AADL_EM émetteur, elle est propagée du composant auquel ce AADL_EM est associé et arrive à l'AADL_EM récepteur. Les transitions AADL déclenchées respectivement par la propagation `out` dans l'émetteur et par la propagation `in` dans le récepteur sont simultanées. La propagation *Sender_Failed* ne cause pas de changement d'état dans l'émetteur. Elle est seulement une conséquence de l'événement interne *Fail* sur l'environnement du composant.

Nous présentons ci-après deux règles possibles pour la transformation de AADL vers RdPSG. Le choix entre ces deux règles dépend du nombre de transitions AADL déclenchées par la propagation (`in` et `out`).

- Cas A : la règle consiste à fusionner la propagation `out` de l'émetteur avec la propagation `in` du récepteur dans une seule transition RdPSG, comme montré dans la Figure 8-a. La propriété d'`Occurrence` de la propagation `out` caractérise le tir de la transition RdPSG.

(a) fusion



(b) découplage

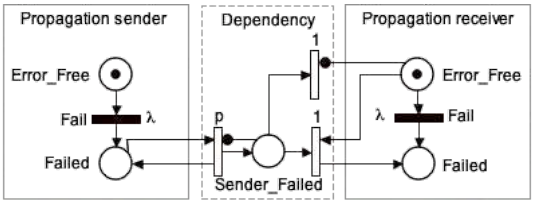


Figure 8 : Propagation de l'émetteur au récepteur

– Cas B : la règle consiste à découpler les propagations `in` et `out` dans le RdPSG par une place intermédiaire, comme montré dans la Figure 8-b. Un jeton arrive dans la place intermédiaire quand la transition RdPSG correspondant à la propagation `out` se produit. Ce jeton est évacué par une transition RdPSG immédiate de probabilité 1 qui a soit i) un arc entrant provenant d'une place correspondant à un état source pour une transition AADL déclenchée par la propagation `in` dans le récepteur, soit ii) des arcs inhibiteurs provenant de toutes les places correspondant à de tels états source. Ainsi, le jeton de cette place intermédiaire est toujours évacué.

Dans le cas le plus général, une propagation `out` déclarée dans un AADL_EM émetteur pourrait déclencher n transitions AADL dans ce même AADL_EM (e.g. une propagation particulière pourrait être propagée à partir de plusieurs états). Des propagations `in` correspondantes pourraient être déclarées dans $r \geq 2$ AADL_EMs récepteurs et pourraient déclencher m_j transitions AADL dans chaque récepteur j ($j = 1 \ldots r$).

Le nombre de transitions RdPSG (N_{tr}) nécessaires pour décrire la propagation AADL est donné par :

Cas A : $$N_{tr} = n * \left(\prod_{j=1}^{r} (m_j + 1) - 1 \right), \forall r \geq 1 \quad [1]$$

Cas B : $$N_{tr} = n + \prod_{j=1}^{r} (m_j + 1), \forall r \geq 1 \quad [2]$$

Le cas B est mieux adapté pour $n=2$ et $m_j > 3$ ($r = 1$) et pour $n>2$ et au moins un $m_j > 2$ (pour $r \geq 2$).

5.3 Systèmes hiérarchiques

Le comportement d'un système en présence de fautes peut être décrit, en utilisant un AADL_EM dérivé, en termes d'états « globaux » dépendant des états des sous-composants. Les AADL_EMs dérivés utilisent des expressions booléennes à l'aide desquelles l'état global du système est déterminé.

Les états globaux du système correspondent à des places dans le RdPSG. L'expression booléenne est transformée en un ensemble de transitions RdPSG immédiates de probabilité 1. Ces transitions sont connectées par des arcs à des places qui correspondent à des états des sous-composants et à des états globaux du système. Seule une place correspondant à un état global doit être marquée à un instant donné. Par conséquent, le nombre de transitions RdPSG correspondant à des expressions booléennes atomiques est égal à n_g-1, n_g étant le nombre de places correspondant à des états globaux. Initialement, un jeton est placé dans la place qui correspond à l'état global initial du système qui est déterminé à partir des états initiaux de ses sous-composants. Chaque transition RdPSG a un arc provenant d'une place correspondant à un état global (qui est vidé quand la transition RdPSG est tirée).

Un exemple est donné dans la Figure 9, qui montre l'implémentation d'un composant AADL de type « système » nommé *A* avec deux sous-composants, *A1* et *A2*.

Figure 9 : Exemple de AADL_EM dérivé

```
system implementation A.nominal
subcomponents
       A1: system hardware.nominal;
       A2: system software.nominal;
annex Error_Model {**
       Model => forA.basic;
       Derived_State_Mapping =>
         Error_Free when
           (A1[Error_Free] and A2[Error_Free]),
         Failed when others; **};
end A.nominal;
```

L'expression `Derived_State_Mapping` spécifie que le système est *Error_Free* si ses deux sous-composants sont *Error_Free*, et *Failed* sinon. La Figure 10 montre le RdPSG obtenu après la transformation du AADL_EM dérivé. La place *Error_Free* correspondant au composant *A* est marquée si les places correspondant à des états *Error_Free* pour *A1* et *A2* sont marquées. Autrement (si au moins une des deux places qui correspondent à des états *Error_Free* pour *A1* et *A2* n'est pas marquée), la place *Failed* de l'AADL_EM dérivé est marquée. Notons que *A* ne peut pas être simultanément dans les états *Error_Free* et *Failed*. Une transition RdPSG correspond à chaque expression booléenne atomique (ici l'expression `when others` est formée de trois expressions booléennes atomiques).

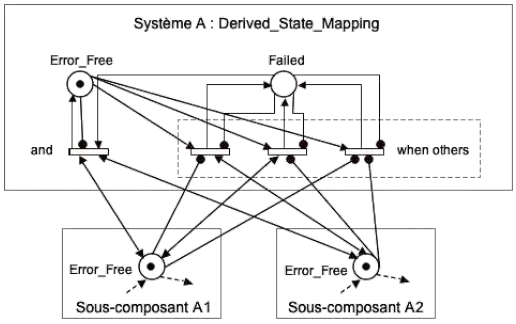


Figure 10 : RdPSG pour l'expression `Derived_State_Mapping`

6 Exemple didactique

Dans cette section, nous illustrons l'approche de modélisation que nous proposons. Nous utilisons les règles de transformation décrites dans la section 5. Pour des raisons de limitation d'espace, nous présentons seulement un exemple simple avec une seule dépendance. Un cas d'étude plus réaliste est présenté dans [18].

Le système considéré ici représente un calculateur simple. Le système est formé de deux sous-composants : un sous-composant matériel (processeur) et un sous-composant logiciel (fil d'exécution) lié au processeur. Des fautes permanentes dans le matériel sont les causes des défaillances du matériel et exigent la maintenance de ce dernier. Des fautes temporaires dans le matériel ne nécessitent pas de maintenance mais peuvent entraîner des propagations d'erreurs au logiciel qui s'exécute au-dessus. Ce comportement représente une dépendance structurelle entre le matériel et le logiciel, car le processeur peut propager des erreurs aux fils d'exécution qui s'exécutent dessus. Le système global fonctionne normalement si le matériel et le logiciel fonctionnent. Sinon, le système est défaillant. Une vue d'ensemble des modèles AADL et RdPSG de sûreté de fonctionnement est donnée dans la Figure 11 en utilisant la notation graphique AADL pour le fil d'exécution, le processeur et la liaison entre le processeur et le fil d'exécution.

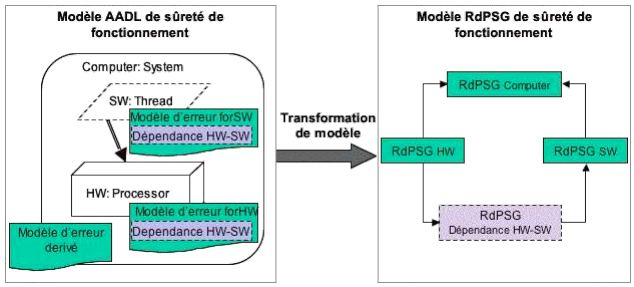


Figure 11 : Vue d'ensemble des modèles

Les deux sous-sections suivantes présentent respectivement la construction de l'AADL_EM et la transformation du modèle AADL en RdPSG.

6.1 AADL_EM du système

La Figure 12 montre à gauche le AADL_EM associé au composant matériel et à droite le AADL_EM associé au composant logiciel.

Le type de l'AADL_EM associé au composant matériel (*forHW)* déclare i) trois états : *HW_Err_Free* (état initial), *HW_Err* et *HW_Failed*, ii) quatre événements : *Temp_Fault*, *Perm_Fault*, *Disappear*, *Repair* et iii) une propagation `out` : *HW_Temp*. L'implémentation de l'AADL_EM associé au composant matériel (*forHW.basic*) déclare des transitions AADL entre les états déclarés dans le type de l'AADL_EM (*forHW*). Si une faute temporaire se produit (*Temp_Fault*), le matériel passe de l'état *HW_Err_Free* à l'état *HW_Err*. Une erreur dans le matériel disparaît (*Disappear*) après un certain temps et le matériel retourne à l'état *HW_Err_Free*. Si une faute permanente se produit (*Perm_Fault*), le matériel passe de l'état *HW_Err_Free* à l'état *HW_Failed*. Une défaillance requiert la réparation (*Repair*) du matériel. Des propriétés d'Occurrence sont associées à tous les événements et propagations `out`. Une erreur causée par une faute temporaire est propagée (*HW_Temp*) avec une probabilité *p1*.

Le type de l'AADL_EM associé au composant logiciel (*forSW*) déclare i) trois états : *SW_Err_Free* (état initial), *SW_Err* et *SW_Failed*, ii) quatre événements : *Fault*, *Recover*, *Non_Recover*, *Restart* et iii) une propagation `in` : *HW_Temp* (correspondant à la propagation `out` du type de l'AADL_EM associé au composant matériel). L'implémentation de l'AADL_EM asocié au composant logiciel (*forSW.basic*) déclare des transitions AADL entre les états déclarés dans le type de l'AADL_EM *forSW*. Quand une faute (*Fault*) se produit, le logiciel se déplace de l'état *SW_Err_Free* à l'état *SW_Err*. Dans certains cas, l'erreur peut être traitée (*Recover*) et le logiciel retourne à l'état *SW_Err_Free*. Dans d'autres cas (*Non_Recover*), le logiciel passe à l'état *SW_Failed* et doit être redémarré (*Restart*). Des propriétés d'Occurrence sont associées à tous les événements. Une propagation `in` causée par une faute temporaire dans le matériel (*HW_Temp*) déclenche une transition AADL de l'état *SW_Err_Free* vers l'état *SW_Err*. L'erreur est ensuite traitée comme si elle était causée par une faute du logiciel.

Le AADL_EM dérivé pour le composant représentant le système global spécifie que le système est en bon fonctionnement quand le matériel et le logiciel sont en bon fonctionnement et il est défaillant dans le cas contraire. L'expression `Derived_State_Mapping` a été décrite dans la Figure 9.

La construction de l'AADL_EM est accomplie en trois itérations : les états et événements (avec des propriétés d'`Occurrence` associées) sont déclarés, avec les transitions correspondantes, dans une première itération. La propagation *HW_Temp* avec sa propriété stochastique et les transitions qu'elle déclenche sont déclarées dans une seconde itération pour expliciter la dépendance entre le matériel et le logiciel (voir la Figure 12). Le AADL_EM dérivé pour le système global est ajouté dans une troisième itération.

6.2 Transformation de modèle

Comme l'étape précédente, la transformation de modèle est effectuée en trois itérations. Le RdPSG résultant est montré dans la Figure 13. Les états et transitions AADL déclenchées par des événements sont transformés dans des places et transitions RdPSG appartenant aux réseaux composants *HW* et *SW*. Les transitions AADL déclenchées par des propagations sont transformées dans des transitions RdPSG qui forment le réseau dépendance *HW-SW*. Le AADL_EM dérivé pour le système global est transformé dans le réseau dépendance dérivé *Computer*.

Pour des raisons de clarté, nous avons modélisé seulement la dépendance structurelle entre les composants matériel et logiciel. Nous avons utilisé la méthode de fusion de la propagation `out` *HW_Temp* déclarée dans le AADL_EM émetteur (*forHW*) avec la propagation `in` *HW_Temp* déclarée dans le AADL_EM récepteur car la propagation `out` *HW_Temp* déclenche une seule transition AADL dans le AADL_EM émetteur (e.g. *n=1*). Nous n'avons pas montré la dépendance liée à la maintenance entre ces deux composants (e.g. le fait que le logiciel ne peut pas être redémarré si le matériel n'est pas en état de bon fonctionnement).

Figure 12 : Modèles d'erreur forHW et forSW

Error Model Type [forHW]

```
error model forHW
features
-- iteration 1
HW_Err_Free: initial error state;
HW_Err, HW_Failed: error state;
Temp_Fault: {Occurrence => poisson λ1};
Perm_Fault : error event
             {Occurrence => poisson λ2};
Disappear: error event
             {Occurrence => poisson μ1};
Repair: error event {Occurrence => poisson μ2};
-- iteration 2 (HW-SW dependency)
HW_Temp: out error propagation
             {Occurrence => fixed p1 };
end forHW;
```

Error Model Implementation [forHW.basic]

```
error model implementation forHW.basic
transitions
-- iteration 1
HW_Err_Free-[Temp_Fault] -> HW_Err;
HW_Err_Free-[Perm_Fault] -> HW_Failed;
HW_Err-[Disappear] -> HW_Err_Free;
HW_Failed-[Repair] -> HW_Err_Free;
-- iteration 2 (HW-SW dependency)
HW_Err-[out HW_Temp] -> HW_Failed;
end forHW.basic;
```

Error Model Type [forSW]

```
error model forSW
features
-- iteration 1
SW_Err_Free: initial error state;
SW_Err, SW_Failed: error state;
Fault: error event {Occurrence => poisson λ3};
Recover: error event {Occurrence => fixed p2};
Non_Recover: error event
             {Occurrence => fixed p3};
Restart: error event
             {Occurrence => poisson μ3};
-- iteration 2 (HW-SW dependency)
HW_Temp: in error propagation;
end forSW;
```

Error Model Implementation [forSW.basic]

```
error model implementation
Receiver_example.basic
transitions
-- iteration 1
SW_Err_Free-[Fault] -> SW_Err;
SW_Err-[Recover] -> SW_Err_Free;
SW_Err-[Non_Recover] -> SW_Failed;
SW_Failed-[Restart] -> SW_Err_Free;
-- iteration 2 (HW-SW dependency)
SW_Err_Free- [in HW_Temp] -> SW_Err;
end forSW.basic;
```

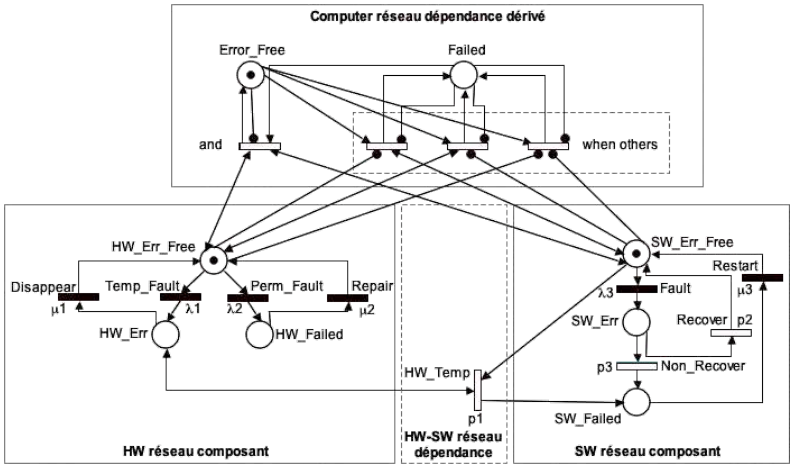


Figure 13 : RdPSG pour le système « computer »

7 Conclusion

Cet article a présenté une approche itérative pour la modélisation de la sûreté de fonctionnement de systèmes informatiques en utilisant le langage AADL comme point de départ et les RdPSG comme formalisme intermédiaire. L'objectif de cette approche est de masquer la complexité des modèles analytiques traditionnels aux utilisateurs qui sont familiarisés avec AADL et qui n'ont pas de connaissances approfondies concernant ces modèles analytiques. Ainsi, nous leur facilitons l'obtention des mesures de sûreté de fonctionnement.

Notre approche vise à assister l'utilisateur dans la construction structurée du modèle AADL de sûreté de fonctionnement (e.g. le modèle d'architecture + les informations liées à la sûreté de fonctionnement). Le modèle AADL de sûreté de fonctionnement est transformé en un RdPSG qui peut être traité par des outils existants. Pour faciliter l'évolution du modèle, nous proposons que le modèle AADL de sûreté de fonctionnement soit construit de manière itérative. Les comportements des composants architecturaux sont d'abord modélisés en présence de leurs propres fautes et événements de réparation. Ensuite, les dépendances entre composants sont modélisées dans les itérations suivantes.

La transformation du modèle AADL en RdPSG est conçue pour être transparente pour l'utilisateur. Par conséquent, elle est basée sur des règles systématiques et rigoureuses, destinées à une mise en œuvre automatique par des outils. La transformation de modèle peut être effectuée de manière itérative, à chaque fois que le modèle AADL de sûreté de fonctionnement est enrichi. Ainsi, la sémantique du RdPSG peut être validé progressivement. Par conséquent, le modèle AADL correspondant peut être aussi validé progressivement et corrigé si nécessaire.

Nous avons illustré l'approche proposée sur un exemple simple avec une seule dépendance. Néanmoins, pour s'assurer de la faisabilité de cette approche, nous l'avons appliquée à un cas d'étude suffisamment complexe dans [18]. Dans cet article, nous avons montré les principes de la transformation et quelques règles. Le travail en cours concerne la finalisation de l'ensemble des règles. Des travaux futurs se concentreront sur l'implémentation d'un outil de transformation de modèle facilement intégrable dans les outils basés sur AADL et RdPSG.

Références

[1] SAE-AS5506, "Architecture Analysis and Design Language," Society of Automotive Engineers, Warrendale, PA 2004.

[2] AADL-WG, "AADL Error Model Annex," soumis au vote formel en Septembre 2005. Des copies peuvent être obtenues par email à info@aadl.info.

[3] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, et I. Mura, "Dependability Modeling and Evaluation of multiple-phased systems, using DEEM," IEEE Transactions on Reliability, vol. 53, pp. 509-522, 2004.

[4] K. Kanoun, M. Borrel, "Fault-tolerant systems dependability. Explicit modeling of hardware and software component interactions," IEEE Transactions on Reliability, vol. 49, pp. 363-376, 2000.

[5] S. Bernadi, A. Bobbio, S. Donatelli, "Petri Nets and Dependability," Lectures on Concurrency and Petri Nets, vol. 3098, W. Reisig and G. Rozenberg ed: Springer-Verlag - LNCS, 2004, pp. 125-179.

[6] C. Béounes, et al., "Surf-2: a program for dependability evaluation of complex hardware and software systems", 23rd IEEE International Symposium on Fault Tolerant Computing, Toulouse, France, pp. 668-673, 1993.

[7] D. D. Deavours, et al., "The Mobius Framework and its Implementation," IEEE Transactions on Software Engineering, vol. 28, pp. 956-969, 2002.

[8] C. Hirel, R. Sahner, X. Zang, and K. Trivedi, "Reliability and performability modeling using SHARPE 2000", 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, Schaumburg, IL, USA, 1789, pp. 345-349, 2000.

[9] S. Bernadi, et al., "GreatSPN in the new millenium", Tool Session of 9th Int. Workshop on Petri Nets and Performance Models, Aachen, Allemagne, 2001.

[10] G. Ciardo, K. S. Trivedi, "SPNP: The Stochastic Petri Net Package (Version 3.1)", 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93), San Diego, CA, USA, pp. 390-391, 1993.

[11] J.-M. Farines, et al., "The Cotre project: rigorous software development for real time systems in avionics", 27th IFAC/IFIP/IEEE Workshop on Real Time Programming Zielona Gora, Pologne, 2003.

[12] OMG, "Unified Modelling Language Specification: version 2.0," http://www.omg.org Octobre 2004.

[13] I. Majzik, A. Bondavalli, "Automatic Dependability Modeling of Systems Described in UML", International Symposium on Software Reliability Engineering (ISSRE), 1998.

[14] A. Bondavalli, et al., "Dependability Analysis in the Early Phases of UML Based System Design," International Journal of Computer Systems - Science & Engineering, vol. 16, pp. 265-275, 2001

[15] G. J. Pai, J. Bechta Dugan, "Automatic Synthesis of Dynamic Fault Trees from UML System Models", 13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, USA, pp. 243-254, 2002.

[16] P. H. Feiler, D. P. Gluch, J. J. Hudak, B. A. Lewis, "Pattern-Based Analysis of an Embedded Real-time System Architecture", 18th IFIP World Computer Congress, ADL Workshop, Toulouse, France, ADL Workshop, pp. 83-91, 2004.

[17] P. Binns, S. Vestal, "Hierarchical composition and abstraction in architecture models", 18th IFIP World Computer Congress, ADL Workshop, Toulouse, France, ADL Workshop, pp. 43-52, 2004.

[18] A. E. Rugina, K. Kanoun, M. Kaâniche, "AADL-based Dependability Modelling," Rapport de Recherche LAAS-CNRS n°06209, April 2006, www.laas.fr/~aerugina/AADLbasedDepModel.pdf.